INFOMOV 2021 EXAM - July 1, 08:45 - 10.45 - RUPPERT-A, D, EDUC-BETA

Answer these questions as elaborate as necessary. Don't be too elaborate: incorrect statements in your answer reduce your score. Negative scores for a question are not possible however.
This exam consists of 5 questions on 2 pages. Your grade is calculated as `(pts*9.f/max_pts)+1`.

1. A few questions on **"Single Instruction, Multiple Data"** a.k.a. SIMD:

a) Explain what 'horizontal' and 'vertical' mean in the context of SIMD processing.
Horizontal: operations that involve multiple lanes. Vertical: operations that stay in their lanes.

How is conditional functionality handled in vectorized code? With masking.

b) For a new CPU design, a designer must choose between 2 cores with 4-wide SIMD support, or 8 cores without SIMD. What might motivate a choice for the 2-core design with SIMD? e.g.: 2 x 4-way SIMD is more constrained than 8 threads but has better data locality and suffers less from false sharing.

2. **On CPU pipelines:**

a) Why do branch mis-predictions lead to latencies on modern CPUs?
The pipeline was filled based on the prediction and must now be refilled.

b) During a working lecture we converted *floating point* code that rotated a particle orb to *fixed point*. The fixed-point version was faster than the floating-point version (by about 10%). Some loop unrolling yielded further gains. However, the fastest option was a *blend*, where each loop iteration transformed one position using fixed point math, and one using the original floating-point math. Explain how this is possible.
Floating point and integer ops run in parallel (instruction level parallelism).

c) Consider the following pseudo-code, which implements the Möller-Trumbore ray/triangle intersection algorithm:

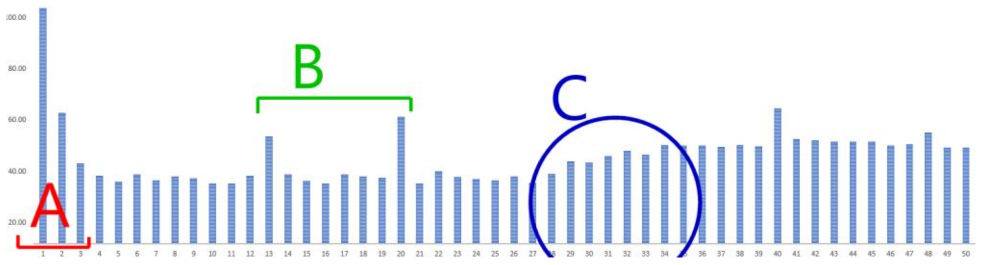
```
h = cross( ray.D, tri.edge2 );
a = dot( tri.edge1, h );
if (a > -0.001 && a < 0.001) return false;
s = ray.O - tri.vertex0;
u = dot( s, h ) / a;
if (u < 0 || u > 1) return false;
q = cross( s, tri.edge1 );
v = dot( ray.D, q ) / a;
if (v < 0 || u + v > 1) return false;
t = dot( tri.edge2, q ) / a;
if (t > 0.001) return false;
return true;
```

CPU: the if-statements cause branch mispredictions, but prevent extra code execution. Personally, I would combine the first two if's and the last two if's; after the first if there is only a dot product but the second if prevents the execution of a cross product.

GPU: threads cannot terminate as on the CPU once a return false is encountered, so doing all the work unconditionally may be best. Then again, this is how the GPU already handles this code… Keeping the if's wil at least allow the GPU to halt a warp for which all threads want to return false.

In this code, four if-statements may lead to termination of the algorithm. An alternative to this flow is *unconditional execution*, perhaps partially. How would you modify this code for CPU to improve speed, and how would you modify it for GPU? Motivate your choice.

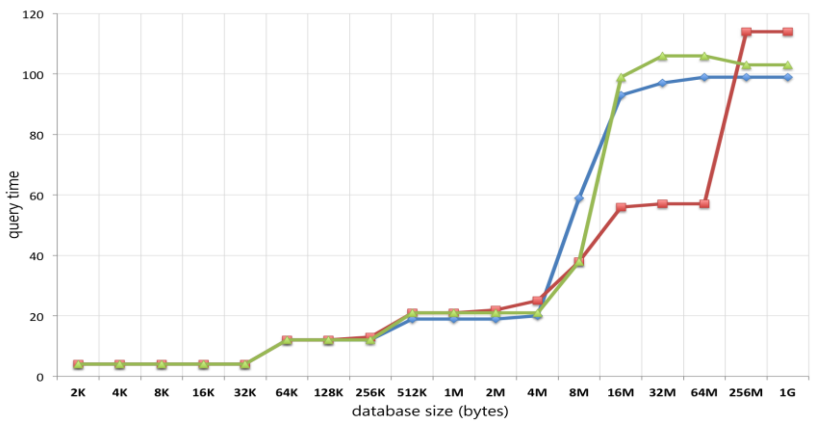
3. **Consider the following profiling graph,** which shows performance of a CPU-based fractal renderer over time, in milliseconds per frame.



Write down plausible causes of the peak at the start (A), the peaks in the mid-section (B) and the increasing frame time after a while (C).

A: caches are 'cold' and need to be filled with the data being used in the code. B: OS-threads or other apps or core migration cause random spikes. C: termal throttling, or perhaps simply zooming in or out on the fractal…

4. The graph below shows the query time for a database, as the dataset grows:



The graph shows results for three different CPUs. Explain what happens.

L1-cache is 32KB. Once data doesn't fit there anymore, we start getting L2 access cost, which is stable until 256KB. Then, L3 kicks in, which is apparently 4MB on these CPUs. Finally, the 'red CPU' has L4 cache, which must be 128MB looking at the graph.

5. On GPGPU:

a) GPUs hide latencies by swapping stalling warps for active warps. However, increasing block size does not always increase performance. Why not?

We need to balance register pressure and occupancy.

b) It is said that GPUs have an *explicit memory hierarchy*, as opposed to the *implicit hierarchy* in CPUs. What do you think this means? Note that these terms may not have been (explicitly) used in the lectures. GPU explicit: shared mem is like L1, but programmer-controlled, so 'explicit'. On the CPU we get (almost) no control over which data is where in the hierarchy.

CUDA allows you to specify how L1 and shared memory is balanced for a particular application running on an Ampere-class NVIDIA GPU (e.g. RTX3080). The available options are:

1. 128 KB L1 + 0 KB Shared Memory
2. 120 KB L1 + 8 KB Shared Memory
3. 112 KB L1 + 16 KB Shared Memory
4. 96 KB L1 + 32 KB Shared Memory
5. 64 KB L1 + 64 KB Shared Memory
6. 28 KB L1 + 100 KB Shared Memory

Option 1: good if we need caching. That means: lots of constant data, and/or lots of textures. All other data does not get cached in L1.

Option 6: good if we explicitly use shared memory. So probably not a good default!

No option 7: looking forward to what you come up with. 😉

c) Describe an application that benefits from option 1.
d) Describe an application that benefits from option 6.
e) Why do you think there is no option 7, with just 128KB shared memory?

6. Consider the following quote: *"Increasingly, application performance is determined by cache effectiveness".*

a) Do you agree, and why/why not?

Say something about the mem/performance gap.

b) How does Data Oriented Design help with this effectiveness? It tends to reduce the number of cache lines we touch, and reduces the amount of unused data. It also improves data locality, decreasing the chance that data gets evicted before we reuse it.

7. And finally: what is the name of this creature, and what does it eat? It's a caracal, and it eats feedback. :)

May the Light be with you! *At all times. See you later!*